Problem A. Adjacent Pairs

Author: Jeroen Op de Beek
Developer: Jeroen Op de Beek
Editorialist: Jeroen Op de Beek

Subtask 1. Let's call an array **good** if all adjacent pairs of elements are different. At the beginning the array is good, and after each operation the array will stay good. So the ending array must have the form: $[c, d, c, d, \dots], c \neq d$. This means that there are $O(n^2)$ possible ending states. (For any ending state with c or d not in $\{1, 2, \dots, n\}$, it doesn't matter what the value is exactly, so there are only $O(n)$ interesting options for those cases).

For this subtask we can try all pairs.

So we fix some c and d. Let's call the final array $\mathbf{b} := [c, d, c, d, \dots]$.

Then we greedily make moves, trying to change elements in $\mathbf{a}$ to $\mathbf{b}$. It could happen that $\mathbf{a} \neq \mathbf{b}$ but no greedy move is possible:

$\mathbf{a} = [1, 2, 3, 1, 2]$

$\mathbf{b} = [1, 2, 1, 2, 1]$

The last three elements of the array cannot be changed greedily, because this would cause adjacent equal valued elements.

Such a blockage is always caused by some subarray that is of the form $[d, c, d, c, ...]$ which is the desired pattern, but with the wrong parities. Let's call subarrays which have values d and c at indices of the wrong parity and that can't be extended further **bad** subarrays. It turns out that changing the second element of any bad subarray to 10^9 is optimal (for the proof, see subtask 2).

So a simple algorithm will try to find a greedy move, if there's no greedy move, it finds any starting point of a bad subarray and changes the second element.

This can be implemented to run in $O(n)$ per iteration, and there are at most $O(n)$ iterations per pair of c and d, so this will run in $O(n^4)$, with a small constant.

Subtask 2. Instead of simulating the process of converting the array we can calculate the number of moves needed more directly. Firstly, for each c and d we find all bad subarrays, with a single for loop.

For a bad subarray of size k it's optimal to first do $\lfloor k/2 \rfloor$ extra moves, where we place the value 10^9 on positions $2, 4, 6, \dots$ of the subarray. After this, the whole subarray can be finished greedily. To show that this is the best we can do, let's consider all possible moves we can make on this subarray.

When a value in the subarray is replaced, we are always left with two bad subarrays of sizes l and r, such that $k = l + r + 1$. Notice that for a bad subarray of size 1, there's no problem. Because it is maximal, the elements around it cannot be equal to c and d, but this means the only element of the subarray can be greedily changed. By induction on the size of the subarray, and basecases 0 and 1 the lowerbound of $\lfloor k/2 \rfloor$ extra moves can be proven. The formula for the number of moves needed will be:

$$\#\text{moves} = n - (\#\text{ of i, such that } a_i = b_i) + \sum_{\text{bad subarrays}} \lfloor k_i/2 \rfloor$$

Now the time per pair is reduced to $O(n)$, and the total complexity is $O(n^3)$

Subtask 3. It's intuitively clear that values that appear more frequent are better candidates for c or d. We can show that instead of $O(n^2)$ pairs, we can only examine the $O(n)$ best pairs. For all pairs c and d, calculate

$$\#\text{greedy moves} = n - (\#\text{ of i, such that } a_i = b_i)$$

and sort them increasingly on this quantity. To calculate this value in $O(1)$ per pair, we can count the number of occurrences of x $(1 \leq x \leq n)$ at odd and even indices as a precomputation step.

Notice that the total number of bad subarrays over all pairs c and d is $O(n)$ (here we ignore bad subarrays of length 1, because they don't change the answer). This is because two bad subarrays cannot overlap by more than 1 element. So there are only $O(n)$ pairs of c and d that cannot immediately be finished by greedy moves. So by the pigeonhole principle, if we examine more pairs than this, we will always have at least one pair with no bad subarrays. After this point no other pairs in the increasing order of greedy moves can give a better answer, so we can break the loop early. With this observation we only need to check $O(n)$ pairs, and the time complexity becomes $O(n^2)$ or $O(n^2 \log(n))$, depending on the implementation.

Subtask 4. Instead of finding bad subarrays for each pair c and d independently, all bad subarrays can be found at once, with a single pass through array **a**, with some simple logic. We can use a map to store $(c, d) \rightarrow$ extra moves needed, and add $\lfloor$bad subarray size$/2\rfloor$ to the map appropriately.

Now for finding the best (c, d) pair, iterate through all possible c, the final value of all odd indices of the array.

For the value of d, we can loop through all d in decreasing order of $\text{occ}_{even}[d]$ (the number of occurrences of d in even positions). By reusing the same observation as subtask 3, we can break this loop over d as soon as (c, d) can be solved with only greedy moves (i.e. it isn't in the map). The total number of iterations of the inner for loop is bounded by (size of the map) $+ n = O(n)$.

This gives an $O(n \log(n))$ algorithm, because of the sorting of d and the map. By changing out the map to a hash map, and changing the sorting algorithm to counting sort, $O(n)$ is also possible. For deterministic linear time, with clever use of a global array that is reused between the iterations of c, the hashmap is no longer needed. These optimizations were not needed to obtain the full score.